HoGent

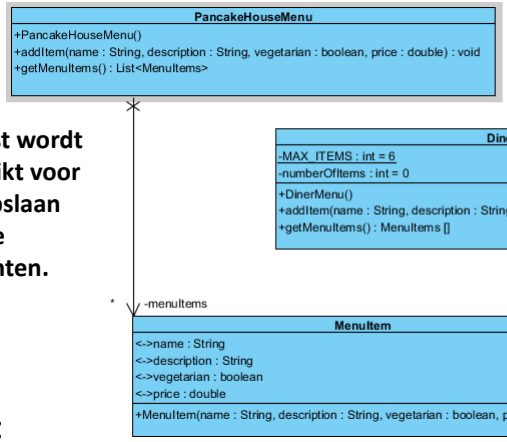
BEDRIJF EN ORGANISATIE

Iterator Pattern – gedrag van objecten
Composite Pattern – structuur van objecten

Fusie van Restaurant Objectville Dinner en Objectville pancake House

De menu-implementaties

- Ze zijn het eens over de implementatie van de gerechten (klasse MenuItem), maar zijn het niet eens over hoe we onze menu's gaan implementeren



Een list wordt gebruikt voor het opslaan van de gerechten.

Een array wordt gebruikt voor het opslaan van de gerechten.

De menu-implementaties

- PancakeHouse


```
public class PancakeHouseMenu {
  private ArrayList<MenuItem> menuItems;
  public PancakeHouseMenu() {
    menuItems = new ArrayList<>();
    addItem("K&B's Pancake Breakfast",
        "Pancakes with scrambled eggs, and toast", true, 2.99);
  //...
  }
  public void addItem(String name, String description,
      boolean vegetarian, double price) {
    MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
    menuItems.add(menuItem);
  }
  public ArrayList<MenuItem> getMenuItems() {
    return menuItems;
  }
```

Opm : we werken voor dit voorbeeld met de concrete implementatie van List, nl ArrayList

De menu-implementatie

- Objectville Diner

```
public class DinerMenu {
  private static final int MAX_ITEMS = 6;  private int numberOfItems = 0;
  private MenuItem[] menuItems;
  public DinerMenu() {
    menuItems = new MenuItem[MAX_ITEMS];
    addItem("Vegetarian BLT", "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99)
    //...
  }
  public void addItem(String name, String description,
      boolean vegetarian, double price) {
    MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
    if (numberOfItems >= MAX_ITEMS) {
      System.err.println("Sorry, menu is full!  Can't add item to menu");
    } else {
      menuItems[numberOfItems] = menuItem;  numberOfItems = numberOfItems + 1;  }
  }
  public MenuItem[] getMenuItems() {
   return menuItems;    }
```

Het probleem met 2 verschillende menu formaten

- We moeten proberen een implementatie te maken voor een client die beide menu's gebruikt.
- We zullen een Java-enabled serveerster maken. De specificatie :
  - printMenu()
    - drukt ieder gerecht op het menu af
  - printBreakfastMenu()
    - drukt alleen ontbijtgerechten af
  - printLunchMenu()
    - drukt alleen lunchgerechten af
  - printVegetarianMenu()
    - drukt alle vegetarische gerechten af
  - isItemVegetarian()
    - levert voor de naam van een gerecht de waarde true op indien het gerecht vegetarisch is, of anders de waarde false.

HoGent

6

Het probleem met 2 verschillende menu formaten

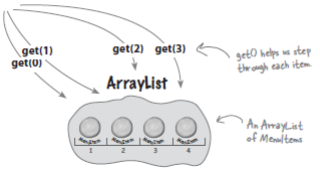
```
public class Waitress {
  private PancakeHouseMenu pancakeHouseMenu;
  private DinerMenu dinerMenu;
  public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
    this.pancakeHouseMenu = pancakeHouseMenu;
    this.dinerMenu = dinerMenu;   }
  public void printMenu() {
    ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();
    for (int i = 0; i < breakfastItems.size(); i++) {
      System.out.print(breakfastItems.get(i).getName() + ", ");
      System.out.print(breakfastItems.get(i).getPrice() + " -- ");
      System.out.println(breakfastItems.get(i).getDescription());      }
    MenuItem[] lunchItems = dinerMenu.getMenuItems();
    for (int i = 0; i < lunchItems.length; i++) {
      System.out.print(lunchItems[i].getName() + ", ");
      System.out.print(lunchItems[i].getPrice() + " -- ");
      System.out.println(lunchItems[i].getDescription());      }
  }
```

Merk op : we maken hier geen gebruik van de ingebouwde iterator/streams in Java, juist om het iterator pattern beter te kunnen doorgronden

Eerste verbetering : Iterator Pattern

- Itereren over ArrayList : size() en get()

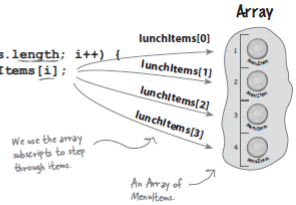
```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
}
```



Merk op : casting is overbodig sinds Java 5

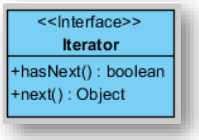
- Itereren over Array : length en subscriptnotatie

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```



Eerste verbetering : Iterator Pattern

- Wat als we de manier van itereren afschermen?
  - Het Iterator Pattern is gebaseerd op een interface met de naam Iterator. Hier volgt een mogelijke Iterator-interface:

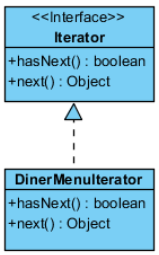



```
public interface Iterator {

   boolean hasNext();
   Object next();

}
```

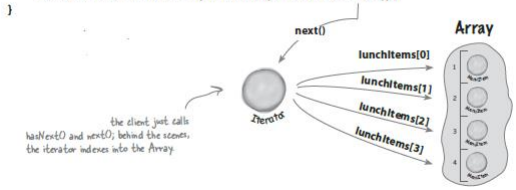
```
Iterator iterator = ...;
while (iterator.hasNext()) {
   Object o = iterator.next();
}
```


Eerste verbetering : Iterator Pattern

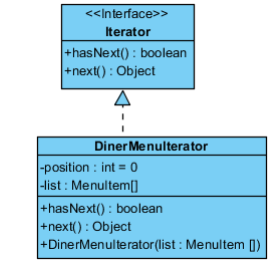
- Wat als we de manier van itereren afschermen?
  - Een implementatie van Iterator weet hoe hij moet itereren door zijn specifieke lijst.
  - DinerMenuIterator is een implementatie van Iterator die weet hoe je moet itereren door een array MenuItems




```
Iterator iterator = lunchMenu.createIterator();
while (iterator.hasNext()) {
    MenuItem menuItem = (MenuItem)iterator.next();
}
```



De iterator : interface - concreet



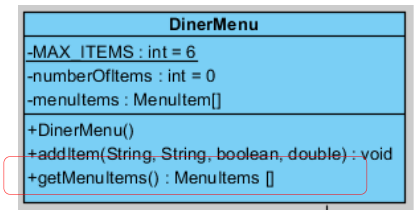

```
public interface Iterator {

   boolean hasNext();
   Object next();

}
```

```
public class DinerMenuIterator implements Iterator {

   private MenuItem[] list;
   private int position = 0;

   public DinerMenuIterator(MenuItem[] list) {
      this.list = list;
   }

   @Override
   public Object next() {
      MenuItem menuItem = list[position];
      position = position + 1;
      return menuItem;
   }

   @Override
   public boolean hasNext() {
      return (position < list.length && list[position] != null);
   }
}
```

HoGent

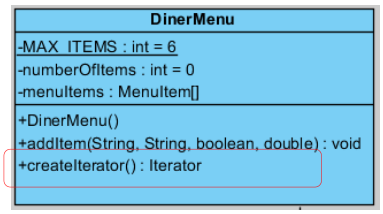
DinerMenu met iterator

- Huidige situatie




```
public MenuItem[] getMenuItems() {
 return menuItems;
}
```

- Met iterator

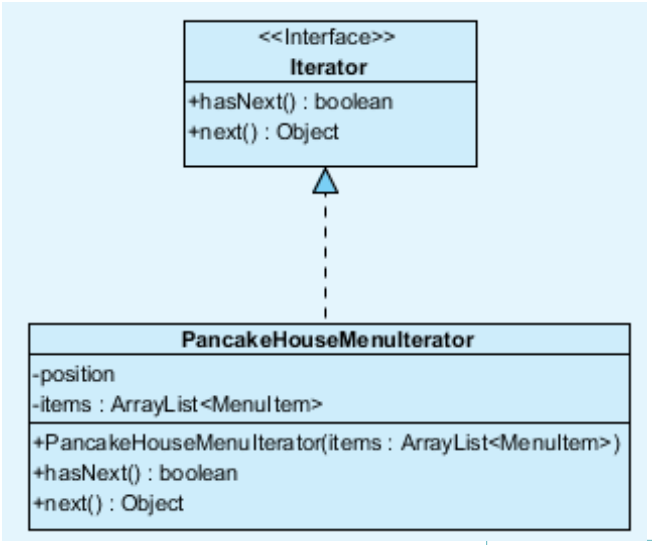



```
public Iterator createIterator() {
 return new DinerMenuIterator(menuItems);
}
```

We gebruiken deze methode niet meer en feitelijk willen we dat niet, omdat deze opgebouwd is op basis van de interne implementatie

HoGent

En nu voor PancakeHouse…



HoGent

Open project IteratorTODO en pas aan

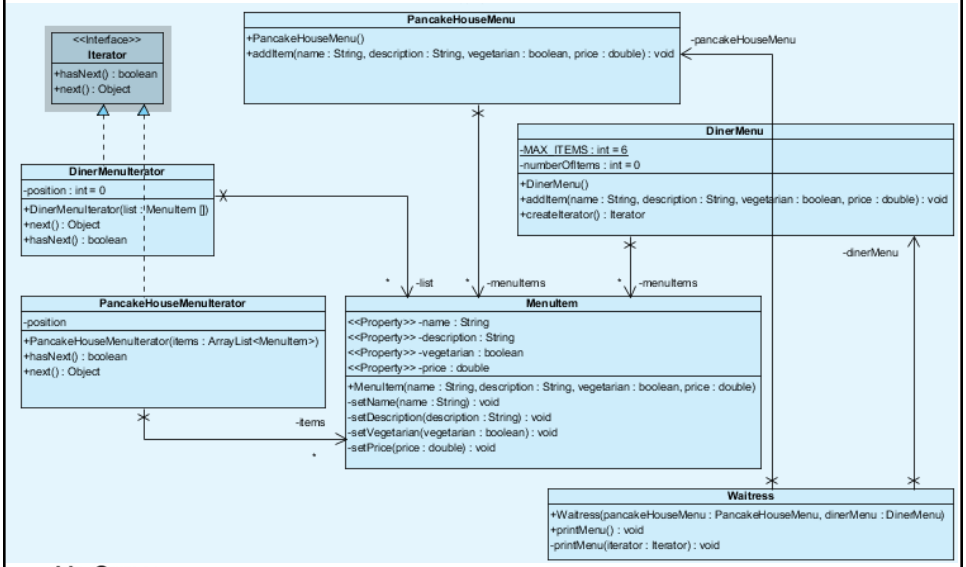
PancakeHouseMenu met iterator

- Pas aan

```
public class PancakeHouseMenu {

    private ArrayList<MenuItem> menuItems;
    public PancakeHouseMenu() {
        menuItems = new ArrayList<>();
        addItem("K&B's Pancake Breakfast",
                "Pancakes with scrambled eggs, and toast", true, 2.99);
        // enz...
    }
    public void addItem(String name, String description,
            boolean vegetarian, double price) {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }
    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }
    // other menu methods here
```

HoGent

De iterator



De klasse Waitress


```
public class Waitress {

  private PancakeHouseMenu pancakeHouseMenu;
  private DinerMenu dinerMenu;

  public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
    this.pancakeHouseMenu = pancakeHouseMenu;
    this.dinerMenu = dinerMenu;
  }

  public void printMenu() {
    Iterator pancakeIterator =                    ;
    Iterator dinerIterator =                      ;
    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);
    System.out.println("\nLUNCH");
    printMenu(dinerIterator);
  }
```

De klasse Waitress

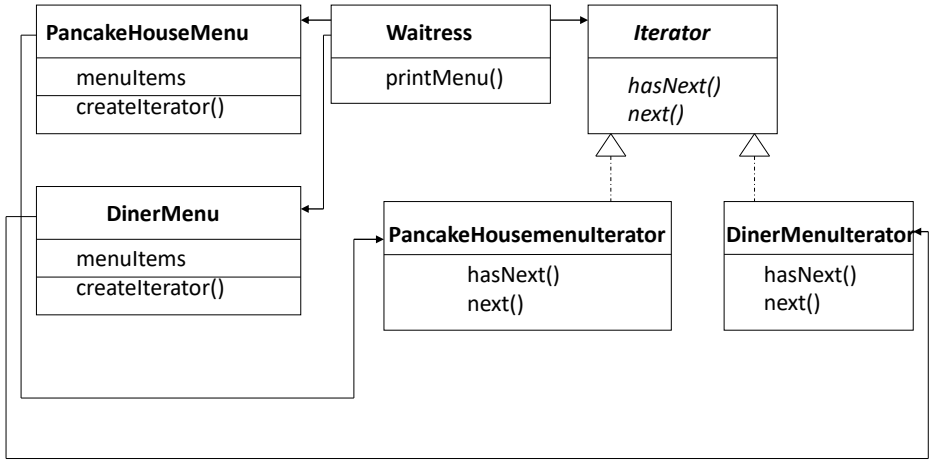

```
private void printMenu(Iterator iterator) {
  while (                                    ) {
    MenuItem menuItem =                                ;
    System.out.print(menuItem.getName() + ", ");
    System.out.print(menuItem.getPrice() + " -- ");
    System.out.println(
        menuItem.getDescription());
  }
}
```


De code testen

```
public static void main(String[] args) {
  PancakeHouseMenu pcm = new PancakeHouseMenu();
  DinerMenu dm = new DinerMenu();

  Waitress waitress = new Waitress(pcm, dm);

  waitress.printMenu();
}
```

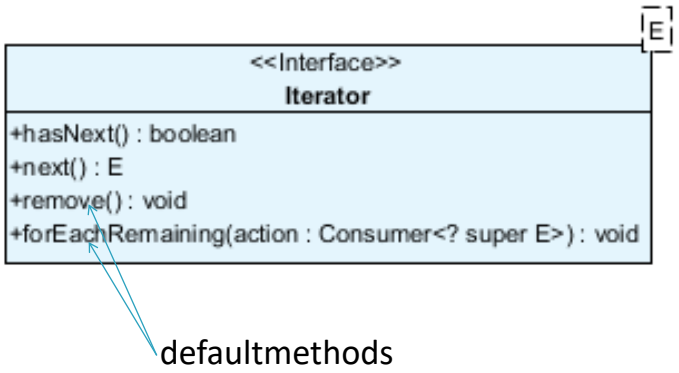
```
Output - Iterator (run)
run:
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries, and blueberry syrup
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
BUILD SUCCESSFUL (total time: 0 seconds)
```

Huidige situatie



Enkele verbeteringen aanbrengen …

- De interface iava.util.Iterator



Opschonen met java.util.Iterator

- De Menu-klassen

```
import java.util.Iterator;
```

  - DinerMenu

```
public Iterator<MenuItem> createIterator()
{
   return new DinerMenuIterator(menuItems);
}
```

  - PancakeHouseMenu

```
public Iterator<MenuItem> createIterator() {
    return  menuItems.iterator();
}
```

Nice! PancakeHouseMenuIterator is overbodig

Opschonen met java.util.Iterator

- De DinerMenuIterator klasse

```
import  java.util.Iterator;
public class DinerMenuIterator implements Iterator {
   private MenuItem[] list;
   private int position = 0;
   public DinerMenuIterator(MenuItem[] list) {...3 lines }
   public Object next() {...5 lines }
   public boolean hasNext() {...6 lines }
   public void remove() {
      if (position <= 0) {
         throw new IllegalStateException("You can't remove an item until you've done at least
      }
      if (list[position - 1] != null) {
         for (int i = position - 1; i < (list.length - 1); i++) {
            list[i] = list[i + 1];
         }
         list[list.length - 1] = null;
      }
   }
```

Opschonen met java.util.Iterator

- De DinerMenuIterator klasse
  - En zelfs deze iterator is overbodig
  - Pas DinerMenu aan :

```
public Iterator<MenuItem> createIterator()
{
   return (Arrays.asList(menuItems)).iterator();
}
```

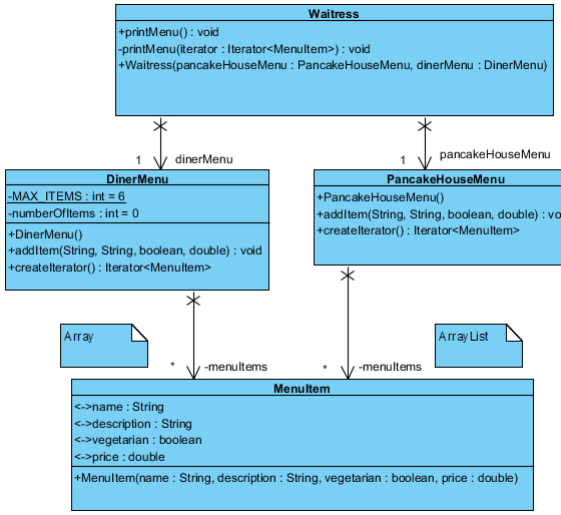

Opschonen met java.util.Iterator

- De Waitress klasse

```
import java.util.Iterator;
public void printMenu() {
   Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
   Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
   System.out.println("MENU\n----\nBREAKFAST");
   printMenu(pancakeIterator);
   System.out.println("\nLUNCH");
   printMenu(dinerIterator);
}

private void printMenu(Iterator<MenuItem> iterator) {
        while (iterator.hasNext()) {
                MenuItem menuItem = iterator.next();
                System.out.print(menuItem.getName() + ", ");
                System.out.print(menuItem.getPrice() + " -- ");
                System.out.println(menuItem.getDescription());
        }
}
```

Opschonen met java.util.Iterator

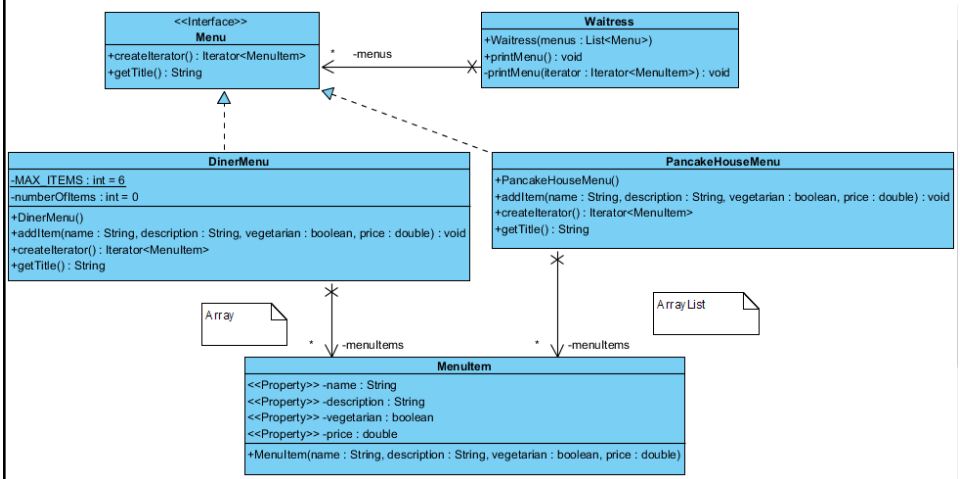
- Het resultaat



Hoe kunnen we het ontwerp nog verbeteren?

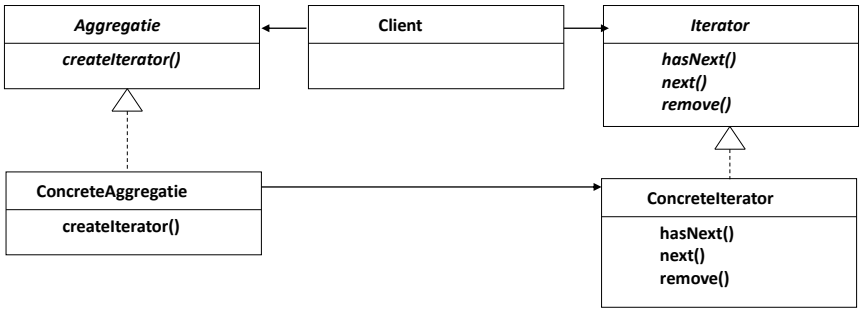
Open project Iterator_java_utilTODO en pas aan

Wat hebben we nu bereikt?



Iterator Pattern

Het Iterator Pattern voorziet ons van een manier voor sequentiële toegang tot de elementen van een aggregaatobject zonder de onderliggende representatie weer te geven.



Eén enkele verantwoordelijkheid

Een klasse dient maar één reden voor verandering te hebben.

- Iedere verantwoordelijkheid van een klasse is een gebied met potentiële verandering. Meer dan een verantwoordelijkheid betekent meer dan een veranderingsgebied. Dit principe geeft aan dat we iedere klasse maar één enkele verantwoordelijkheid moeten geven.

Fusie met Café Menu

Hoe integreren in het framework?

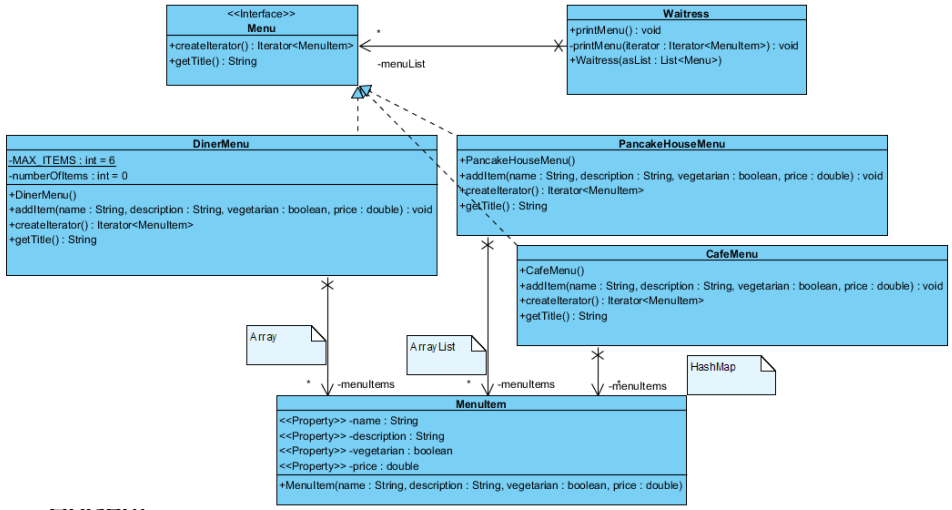
- Een blik op het Café Menu

```
public class CafeMenu implements Menu {
  private Map<String, MenuItem> menuItems = new HashMap<>();

  public CafeMenu() {
    addItem("Veggie Burger", "Veggie burger with salad", true, 3.99);
    addItem("Soup of the day", "Soup with bread", false, 3.69);
    addItem("Burrito", "Burrito", true, 4.29);
  }

  public void addItem(String name, String description, boolean vegetarian,
      double price) {
    MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
    menuItems.put(menuItem.getName(), menuItem);
  }

  public Map<String, MenuItem> getMenuItems() {
    return menuItems;
  }
```

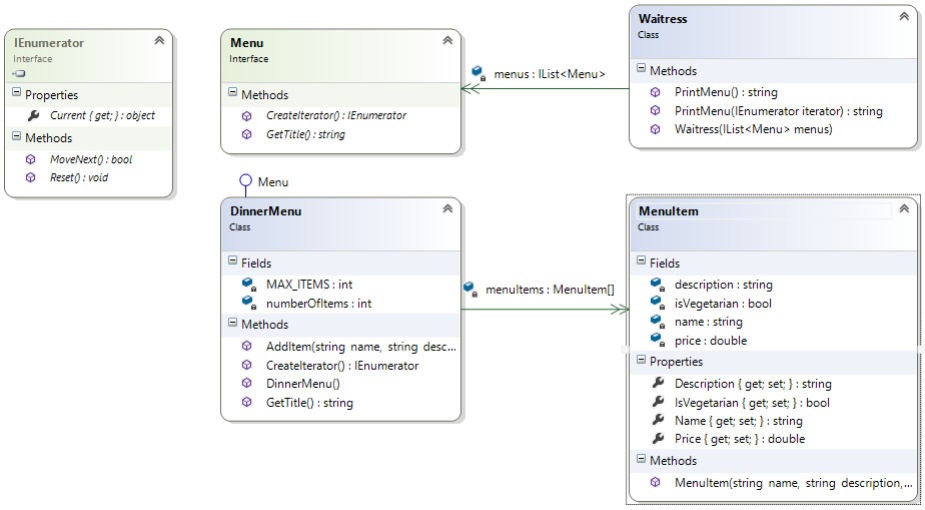
Ho

Café menu integreren in framework

Pas aan in project Iterator_java_utilTODO



Voorbeeld in .Net



Voorbeeld in .Net

```
public class Waitress
{
   private IList<Menu> menus;

   public Waitress(IList<Menu> menus)
   {
      this.menus = menus;
   }

   public string PrintMenu()
   {
      StringBuilder sb = new StringBuilder();

      foreach (Menu m in menus)
      {
         sb.Append(m.GetTitle());
         sb.Append(PrintMenu(m.CreateIterator()));
      }
      return sb.ToString();
   }
```

```
public string PrintMenu(IEnumerator iterator)
{
   StringBuilder sb = new StringBuilder();
   while(iterator.MoveNext())
   {
      MenuItem menuItem = (MenuItem)iterator.Current;
      sb.Append(menuItem.Name + ", ");
      sb.Append(menuItem.Price + " -- ");
      sb.Append(menuItem.Description + "\n");
   }
   return sb.ToString();
}
```

```
public interface Menu
{
   IEnumerator CreateIterator();
   string GetTitle();
}
```

DinnerMenu

```
public IEnumerator CreateIterator()
{
   return menuItems.GetEnumerator();
}
```

Java8 Streams & iterator

- Iterators kunnen in twee soorten opgedeeld worden: Active en Passive.

- Active iterator (ook expliciet of external iterator)
  - De client creëert en bestuurt de iterator (for loops of iterators).
- Passive iterator (ook impliciet, internal of callback iterator)
  - De client geeft de opdracht aan de iterator, de iterator bestuurt zichzelf.
  - Streams gebruiken passive iterators.
  - **Iterable** interface voorziet een default methode **foreach**()
  - **forEach**() is geïmplementeerd door een active iterator.

En net toen we dachten dat alles veilig was …

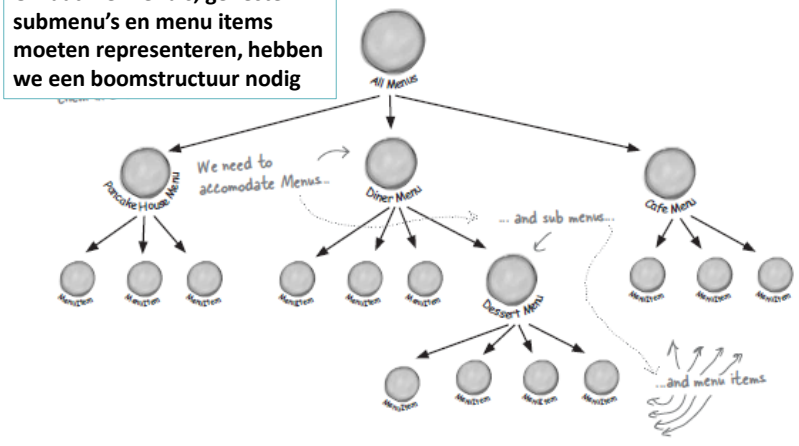
Diner Menu moet een submenu kunnen bevatten, maar we kunnen geen menu aan de array MenuItem toekennen omdat de typen verschillend zijn.

We kunnen aan een MenuItem-array geen dessertmenu toekennen.

Tijd voor verandering!

Wat hebben we nodig?

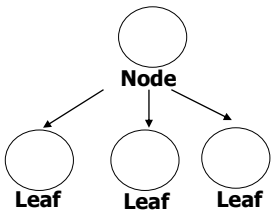
Omdat we menu's, geneste submenu's en menu items moeten representeren, hebben we een boomstructuur nodig

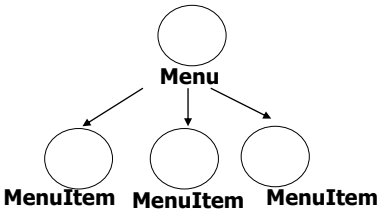


- Hoe zou jij met deze nieuwe toegevoegde ontwerpeisen omgaan?

Composite Pattern

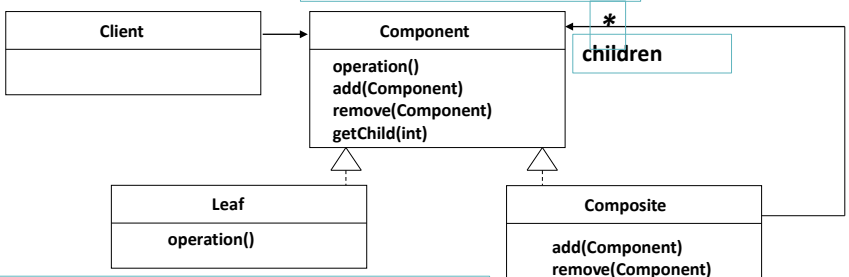
Het Composite Pattern stelt je in staat om objecten in boomstructuren samen te stellen om partwhole hiërarchiën weer te geven. Composite laat clients de afzonderlijke objecten of samengestelde objecten op uniforme wijze behandelen.





Composite Pattern

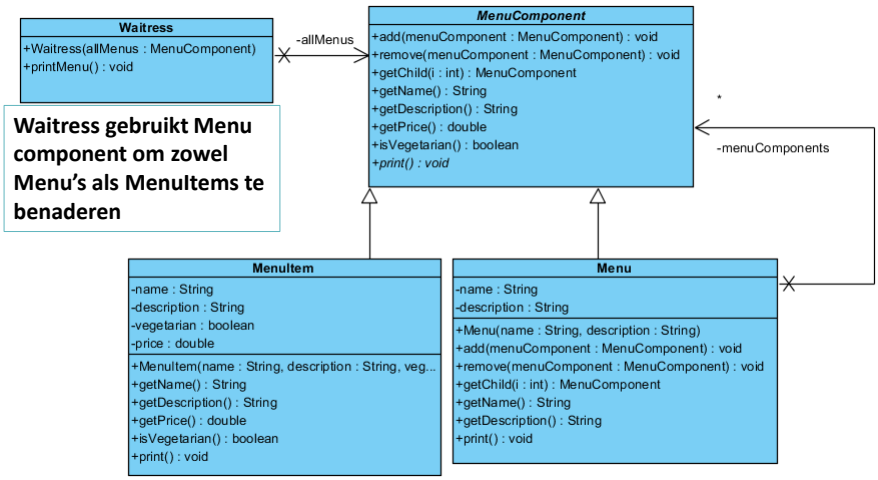
Definieert een interface voor alle objecten in de compositie : voor zowel de samengestelde als de bladeren



- Definieert het gedrag voor de elementen in de compositie. Dit gebeurt via de implementatie van de operaties die Composite ondersteunt.
- Een blad heeft geen kinderen
- Merk op : Leaf erft ook add, remove, getChild wat niet logisch lijkt! Hier komen we straks op terug

- Definieert het gedrag van de componenten met kinderen.
- Hij moet ook de kinderen bijhouden

Menus ontwerpen met Composite



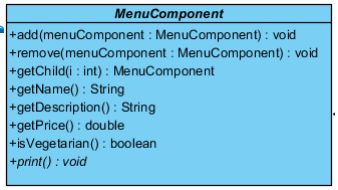
Waitress gebruikt Menu component om zowel Menu's als MenuItems te benaderen

Zowel Menu als MenuItem overriden enkel de methoden die zin hebben, en gebruiken de standaardimplementatie uit MenuComponent voor de methoden die niet zinnig zijn

HoGent

Implementatie van Menu

```
public abstract class MenuComponent {
//compositie-methoden
  public void add(MenuComponent menuComponent) {
    throw new UnsupportedOperationException();
  }
  public void remove(MenuComponent menuComponent) {
    throw new UnsupportedOperationException();
  }
  public MenuComponent getChild(int i) {
    throw new UnsupportedOperationException();
  }
  //operatie methoden gebruikt door MenuItem, sommigen ook door Menu
  public String getName() {
    throw new UnsupportedOperationException();
  }
  public String getDescription() {
    throw new UnsupportedOperationException();
  }
   public double getPrice() {
     throw new UnsupportedOperationException();
   }
   public boolean isVegetarian() {
     throw new UnsupportedOperationException();
   }
   public abstract void print();
}
```

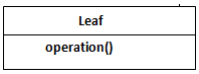


Alle componenten moeten de MenuComponent interface implementeren. Omdat bladeren en knopen een verschillende rol hebben, kunnen we niet altijd een standaard implementatie voor iedere methode definiëren die zinnig is. Soms is het beste het throwen van een Exception

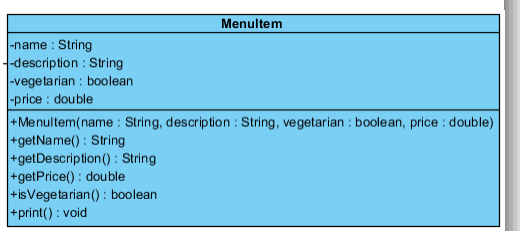
Omdat enkele van deze methoden logisch zijn MenuItem en anderen alleen voor Menu, is de standaardimplementatie UnsupportedException. Ondersteunt MenuItem of Menu een operatie niet, dan hoeven ze niets te doen. Ze erven gewoon de standaardimplementatie

39

Implementatie van MenuItem




```
public class MenuItem extends MenuComponent {

  private String name;   private String description;
  private boolean vegetarian;   private double price;
  public MenuItem(String name,  String description, boolean vegetarian, double price) {
    this.name = name; this.description = description;
    this.vegetarian = vegetarian; this.price = price;
  }
  public String getName() {
    return name;
  }
  public String getDescription() {
    return description;
  }
  public double getPrice() {
    return price;
  }
```



HoGent

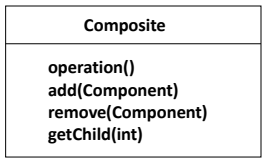
40

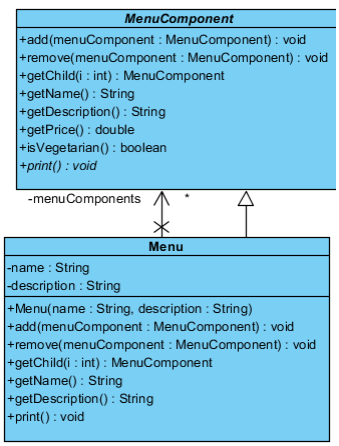
Implementatie van MenuItem(vervolg)

```
  public boolean isVegetarian() {
    return vegetarian;
  }
  public void print() {
    System.out.print("  " + getName());
    if (isVegetarian()) {
      System.out.print("(v)");
    }
    System.out.println(", " + getPrice());
    System.out.println("    -- " + getDescription());
  }
}
```


Implementatie van Menu

```
public class Menu extends MenuComponent {

  private List<MenuComponent> menuComponents = new ArrayList<>();
  private String name;
  private String description;

  public Menu(String name, String description) {
    this.name = name;
    this.description = description;
  }

  public void add(MenuComponent menuComponent) {
    menuComponents.add(menuComponent);
  }

  public void remove(MenuComponent menuComponent) {
    menuComponents.remove(menuComponent);
  }

  public MenuComponent getChild(int i) {
    return menuComponents.get(i);
  }
```





Implementatie van Menu (vervolg)

```
public String getName() {
   return name;
}

public String getDescription() {
   return description;
}

public void print() {
   System.out.print("\n" + getName());
   System.out.println(", " + getDescription());
   System.out.println("----------------------");

   menuComponents.forEach(MenuComponent::print);
   // Aangezien zowel Menu als MenuItem print() implementeren hoeven we
   // alleen print() aan te roepen.
}
```

Merk op : we doen geen override van getPrice en isVegetarian omdat deze methoden voor een Menu zinloos zijn

Om het Menu af te drukken, drukken we de naam en beschrijving af, maar moeten we ook de info over alle componenten van Menu, dus andere Menu's en MenuItems afdrukken

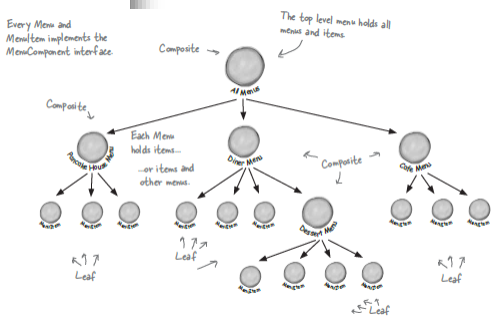
Opm : als we tijdens deze iteratie een ander Menu object tegenkomen, dan zal de methode print() een andere iteratie starten,…. RECURSIE!

De Waitress

```
public class Waitress {

   private MenuComponent allMenus;

   public Waitress(MenuComponent allMenus) {
      this.allMenus = allMenus;
   }

   public void printMenu() {
      allMenus.print();
   }
```

Wauw. De code is echt eenvoudig. We geven de waitress het toplevelmenucomponentje, di het componentje dat alle andere menu's bevat. We hebben deze allMenus genoemd

Het enige dat ze hoeft te doen om de menuhiërarchie af te drukken, dus alle menu's en alle menu-items, is het aanroepen van het toplevelmenu



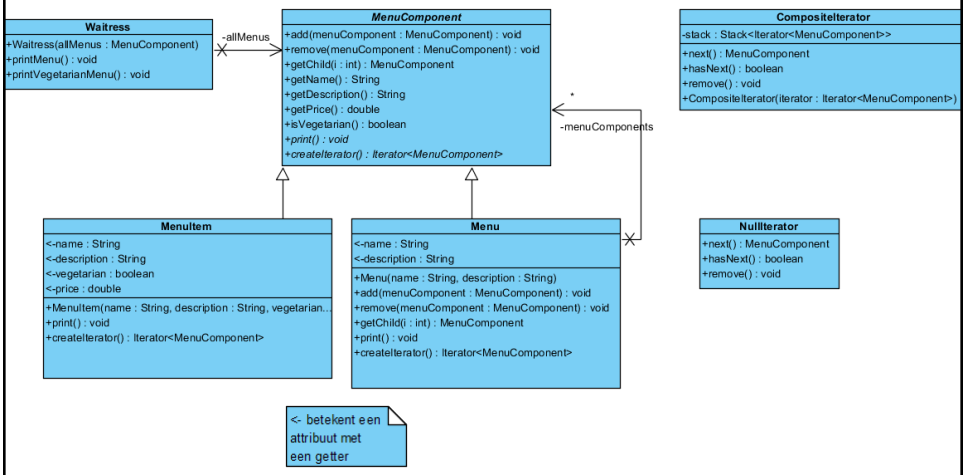
De Testrun

- Zie klasse CompositeStartUp

Een iterator samen met Compositie

- We hebben een Iterator in onze interne implementatie van de methode print() gebruikt, maar we kunnen Waitress ook toestaan om over de hele compositie te itereren als dat nodig is. Bv. als ze het hele menu wil doorlopen en daar de vegetarische gerechten uit wil halen.
- Om een Composite-iterator te implementeren voegen we de methode createIterator() toe aan iedere component.

De compositie-iterator



Terugblik op de iterator

MenuComponent
+add(menuComponent : MenuComponent) : void
+remove(menuComponent : MenuComponent) : void
+getChild(i : int) : MenuComponent
+getName() : String
+getDescription() : String
+getPrice() : double
+isVegetarian() : boolean
+print() : void
+createIterator() : Iterator<MenuComponent>

- In Menu

```
@Override
public Iterator<MenuComponent> createIterator() {
  return menuComponents.iterator();
}
```

  - Een knoop geeft een iterator terug over zijn kinderen.
  - Als we over de hele boomstructuur wensen te itereren dienen we de iterator van de root in een CompositeIterator te verpakken.
- In MenuItem

```
@Override
public Iterator<MenuComponent> createIterator() {
  return new NullIterator();
}
```

  - Oei? Moeten we hier kunnen itereren. Is dit zinvol? Wat doet de NullIterator?

De compositie-iterator

```
public class CompositeIterator implements Iterator<MenuComponent> {
  private Stack<Iterator<MenuComponent>> stack = new Stack<>();
  //parameter : de iterator van de rootcompositie waardoor we gaan itereren
  public CompositeIterator(Iterator<MenuComponent> iterator) {
    stack.push(iterator);
  }

  public MenuComponent next() {
    if (hasNext()) { //Is er wel een volgend element?
      //haal lopende iterator van de stack en ga naar volgend element
      Iterator<MenuComponent> iterator = stack.peek();
      MenuComponent component = iterator.next();
      //Is dit element een Menu, dan moet die compositie in de iteratie worden o
      //Dus zetten we het op de stack (en zullen we sowieso later verwerken)
      if (component instanceof Menu) {
        stack.push(component.createIterator());
      }
      return component;
    } else {   return null;   } } // er isgeen volgend element
```


De compositie-iterator

```
@Override
public boolean hasNext() {
  if (stack.empty()) { //is er een volgend element? Niet als de stack leeg is
    return false;
  } else {
    //Haal iterator van de top van de stack
    Iterator<MenuComponent> iterator = stack.peek();
    if (!iterator.hasNext()) { //en kijk of er een volgend element is
      //zo niet, dan poppen we hem van de stack en roepen hasNext() recursief aan
      stack.pop();
      return hasNext();
    } else { //er is een volgend element
      return true;
    }
  }
 }
}
```

De null iterator

- Optie 1 : return null
  - Maar dan conditionele code nodig in de client
- Optie 2 : NullIterator
  - Retourneert altijd false bij aanroep van hasNext()!

```
public class NullIterator implements Iterator<MenuComponent> {
    @Override
    public MenuComponent next() {
        return null;
    }
    @Override
    public boolean hasNext() {
        return false;
    }
}
```


Geef me een vegetarisch menu

- Klasse Waitress

```
public void printVegetarianMenu() {
    //We wensen over de hele boomstructuur te itereren, we verpakken de iterator van de root
    //in een CompositeIterator die weet hoe hij een willekeurige compositie moet itereren
    Iterator<MenuComponent> iterator = new CompositeIterator(allMenus.createIterator());
    System.out.println("\nVEGETARIAN MENU\n----");
    while (iterator.hasNext()) {
        MenuComponent menuComponent = iterator.next();
        try {
            if (menuComponent.isVegetarian()) {
                menuComponent.print();
            }
        } catch (UnsupportedOperationException e) {
        }
    }
}
```